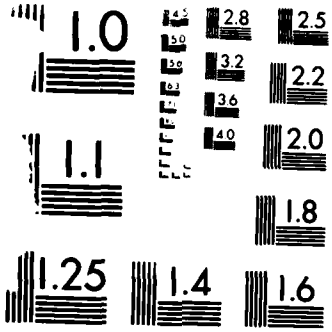
AD-A165 335 TEACHING ADA: A COMPARISON OF TWO APROACHES(U) MARYLAND 1/1
UNIV COLLEGE PARK J BAILEY 1984 N00014-82-K-0225

UNCLASSIFIED F/G 5/9 NL



Teaching Ada: A Comparison of Two Approaches

John Bailey
Software Metrics, Inc.

N00014-82-K-0225

Abstract

This paper describes an experimental comparison of two approaches to teaching the Ada language. The goal was to discover an effective way to teach students the use of the language as a vehicle to apply information hiding and data abstraction to software development. The fifty-four participants in the study were enrolled in an advanced undergraduate Ada class at the University of Maryland, MD. Baseline data was gathered on every student, including programming aptitude scores. The class was then randomly divided into two sections. One section was taught the language features first, approximately in the order that they are presented in the language manual, and then shown how packages can be used to encapsulate objects, resources, and types when a system is first designed. The other section was taught these principles of encapsulation first by learning to use the Ada package to express designs before the lower-level language features were presented. The same set of lectures was eventually presented to both sections.

Although it was initially hypothesized that the section which learned design first would ultimately produce more modifiable programs, the lack of complete, executable examples during the entire first half of the course appeared to hamper a complete understanding of the concepts. Ultimately, the high variability among the students masked any large differences between the sections. However, some interesting differences in the correlations between background data and the success of the students in each section were revealed. Although this experience suggests that a better way to teach Ada might be to combine the two approaches attempted here by presenting complete examples while continually emphasizing design considerations, the optimal approach would probably involve tailoring a curriculum to each student's background and experience.

Introduction

The software development industry is faced with the task of educating thousands of programmers in Ada. We have already seen that an education in the language syntax and semantics, even one which illustrates proper use on small examples, is not sufficient to enable industry programmers to apply that knowledge to derive well-encapsulated solutions to large problems [1, 2]. An education in the effective use of Ada must contain substantially more than direction in how to code solutions in the new syntax. It must contain direction and insight into the design process and how the language can be used to support its goals of maintainability, reliability, and reusability when dealing with large systems.

Since Ada was adapted as a standard before any complete translator was available, no widespread experience in learning to use the new language was available during its review. Yet, a goal of the Ada language has been to improve communication among humans. Mechanical translation speed was secondary to the desire to make Ada readable and communicative. The required experience using and teaching Ada will take

acquire, but efforts such as this one to document these processes must begin. In the near term, this will aid our understanding of how to best teach the language, and eventually it will be needed as input to future revisions of the language (which are currently expected near the end of the decade).

In an effort to broaden our understanding and skills in the area of education, this study examined the effectiveness of two approaches to teaching Ada as an advanced software engineering language. Although there were several attributes of such a curriculum which seemed desirable and would warrant support from experimental evidence, this study examined only one. This was to ensure the localization of the source of any differences found. However, a broad selection of data was gathered on each student before, during, and after the course which has proven particularly valuable by providing additional insight and allowing for some interesting conclusions about the process of learning to use Ada.

Approach

The general hypothesis was that a more useful learning would be accomplished by the students in a class which used Ada as a vehicle to express good program design and structure than by students who learned good program design and structure as one aspect to learning Ada. In the first case, the principles would be taught first and the details of Ada later, while in the second case the order would be the opposite. The difference appears subtle, however the latter approach was taken in the training of the programmers for the 1982 General Electric Ada pilot development [3]. The problems encountered and the reflections of the programmers as they discovered the shortcomings of their design pointed out the fact that an initial emphasis on the goals of the proper use of Ada and the use of packaging to achieve those design goals would have been valuable. In the General Electric example, it was suggested that preceeding (instead of following) a language course with a methodology course would have made the proper use of Ada's features clearer and ultimately more meaningful. This philosophy is reminiscent of the motivation behind "new math" which appeared in the grade schools here more than twenty years ago.

Independent Variable

The variable for experimental comparison, then, was the order of introduction of the topics of methodology and syntax. Since the higher level features of Ada (packaging, separate compilation) are more closely associated with design and methodology, it was decided that those language features should be introduced first in the class which learned good program design and structure first. The class which learned the mechanics of the language first would learn the features in a more conventional, bottom-up manner. The experimental comparison was between two sections of a course, one with the topics taught in a conventional, building block, analytical order, and the other taught in essentially the reverse order.

The conventional class, which was taught by starting with syntax, can be thought of as the control class for this study. To lend credibility to this order, this section followed approximately the order used in the language reference manual [4] and in the Ada text by co-designer John Barnes [5]. The other class, the experimental class for this study, was taught the principles of modularity, information hiding, and data abstraction first and the Ada syntax which supports these later, starting with the most outer-level structuring features of the language. It is important to note that the time devoted to each topic (syntax vs. motivation) was the same for both classes. The only variable was the order of the topics.

There are several arguments to support the approach used in the experimental class. For one, the "outside-in" order of the features is the order with which they are applied in a systematic decomposition of a problem into design and eventually executable code. The use of Ada as a design language evolves naturally this way. In this way, the tools to make the most crucial design and methodological decisions would be acquired first. Also, it was expected that by teaching more foreign concepts first, students would be less apt to attempt to learn Ada in some previously understood context, such as Pascal or even FORTRAN. It was hoped that this would help avoid functional fixedness [6] and negative transfer [7] which could lead to using the language as a transliteration of previously mastered languages.

Correlational Study

Since the difference between the two classes might be hard to demonstrate because of the large individual differences that are always found in experiments such as this [8], a back-up analysis was anticipated which would exploit these differences by correlating aptitude and past experience with success in the course. This seemed particularly sensible since the same individual differences which would hamper the comparison between sections would probably

contribute to this aspect of the study. This alternate goal involves the analysis of the effects of these individual differences on the ease of learning new program design concepts. On the General Electric project which used Ada as a design and code language, the application of the concept of data abstraction was apparently clearest to the programmers who had diverse experience and most foreign to the programmers with only FORTRAN backgrounds [1]. This was independent of length of experience. A higher adaptability to new concepts by programmers with broader (but not necessarily longer) experience has also been shown through other studies at General Electric which examined programmer performance on coding, comprehension, and modification tasks [9]. This would seem to indicate that there are certain skills acquired through breadth of experience which allow for a level of comprehension of a new language which exceeds the mechanical application of the syntax and semantics of that language.

If there are new cognitive skills associated with the complete acquisition of a new language, then might these skills in turn be useful in the functions of thought, expression, and problem solving? The hypothesis that the particular language used affects one's ability to think about a situation or solution is not unique to computer languages. Although some debate exists on the hypothesis, Whorf [10] proposed that natural language restricts and determines thought. For example, English-speakers will interpret the sentence, "The stone falls." by determining that an object and its motion is significant. In this case, falling is an operation that objects in our experience can perform. It seems odd to interpret such a thought any other way. Yet speakers of Nootka (a native American language) have no word similar to the verb "to fall." Instead, the closest sentence is one which describes a "stoning action" and translates roughly to "It stones down." This as an action based description while the English version seems object based, which suggests that the customary way to perceive a situation differs depending on the language context of the observer.

A similar situation may be a factor in learning and using computer languages. For decades, FORTRAN has been used to divide problems into composite actions. New approaches to designing which are only beginning to receive attention in industry suggest that dividing problems into typed objects and resources which can accomplish or display actions will lead to programs which are more maintainable and reusable because they can be composed of independent, well defined modules. (Although "object oriented" has been used to describe this style of design, the term is avoided here because of the debate over its several connotations.) Ada is probably the first language to directly support this style of designing which will enjoy widespread use in industry. Because of the patterns of thought which are probably familiar to the veteran FORTRAN programmer, it is possible that this style of use of the language will be difficult at first for these individuals.

Other issues from cognitive psychology which might be related to the acquisition of a new language are those of functional fixedness, where a certain element in one context is not seen to possess other capabilities in another context, and negative transfer, where certain prior knowledge is actually detrimental to new insights. An example of functional fixedness is the familiar brain-teaser where one imagines he is in a room with two strings hanging from the ceiling which are to be tied together. Unfortunately, they are too far apart to reach simultaneously. A person who is not susceptible to functional fixedness may solve the problem by tying a pair of pliers to one string and causing it to swing so it can be caught while holding the other string. The fact that the pliers are intended for other purposes, however, will prevent some from reaching this solution. With respect to programming language features, some features may have different purposes in different contexts. One example is a procedure. In algorithmic abstraction, the procedure is principally an encapsulation of some processing steps. However, in a data abstraction, a procedure is seen as the access to a resource or object. Or, in a message-object design, a procedure call is thought of as a message. Although, in detail, the feature serves the same purpose, at the high level, its function is seen differently. A case could be made for the effects of this negative transfer if students with a great deal of experience entirely in algorithmic abstraction have more difficulty switching over to object abstraction than students will less experience.

Data Collection and Dependent Variables

The measurements of each students' work were designed to track their progress in acquiring a facility with design concepts, and specifically data and object abstraction. These were in the form of quizzes, tests, assignments, and a final exam. Short (five-minute) quizzes were given at the conclusion of most class meetings to examine the effectiveness of the lecture and to get an initial measurement from each student on each topic. Two assignments and two tests were designed to measure the integration of topics taught over the semester and were similar between the sections (but not

identical to discourage inter-section communication by the students). The final exam was identical for both sections.

Other measures were derived from subject profiles which were compiled from the results of a background questionnaire (experience, courses, grades, interests), a glossary questionnaire, and a standardized programming aptitude test from Science Research Associates, Inc., known as the Computer Programmer Aptitude Battery. None of this material was examined or scored until the completion of the course to prevent any possibility of bias from this knowledge on the part of the author.

Confoundings

Probably the most obvious source of criticism for the design was that the author taught both sections and could have adversely affected the quality of instruction to the control section. To help defend the equality of instruction provided, video recordings were made of all classes to ensure (and demonstrate) that the same general lecture was given to the other section later in the semester. Also, a significant counter argument which favors the control section is that the crucial instruction, that which was intended to be detectable by the dependent variables, was given first in the experimental section. Thus, there might have been a learning effect on the part of the lecturer so that by the second time they were presented, the control section actually received better instruction on the key topics.

Other factors were eliminated or randomized as much as possible. For example, random assignment of all registrants to the two sections was performed and all classes met at the same time of day in the same room and at the same intervals.

Data

The data for each student were condensed into eleven variables. Seven of these describe each student prior to taking the course, three more were collected during the course, and one (part of the final exam) was taken immediately after the course. A much larger volume of raw data is available, but the chosen factors represent a selection and condensation which were immediately available and judged to be representative, straightforward, and useful.

Background Data

The background data items were grade point average, SAT verbal and non-verbal scores, a standardized programmer aptitude test score, two scores from a computer science glossary questionnaire, and a score derived from a general background questionnaire. The grade point average was computed from all courses taken at the University of Maryland, and not only computer science courses. The SAT scores were from late high school and were available for all but six of the fifty-four students. The Computer Programmer Aptitude Battery (CPAB) was purchased from Science Research Associates, Inc. to obtain a score for industry-related programming ability. The test takes seventy-five minutes and measures verbal ability, reasoning, letter series ability, number ability, and diagramming. It is recommended by SRA for applicant selection and placement and has been shown to be a fair predictor of job success at several independent companies. Further discussion of this test can be found in [11].

The glossary questionnaire developed for this study was designed to determine each student's knowledge of a variety of terms. Fourty-eight terms were presented with instructions to define all recognized terms in the context of computer science. Several ambiguous terms were included to determine which of two or more possible connotations each student had for each. The purpose of including the ambiguous terms was to reveal whether a student had a predisposition to software engineering concepts or a predisposition to computer hardware and operation. Two scores were computed from the responses to the eight ambiguous terms on the list, one associated with software engineering concepts and the other with computer hardware and operation concepts. (The remainder of the questionnaire was not scored at this time.) The selected terms were "object," "type," "module," "Smalltalk," "process," "virtual machine," "interface," and "abstraction." Each of these terms may have multiple connotations with at least one of these indicating a knowledge of or an orientation toward concepts within the realm of software engineering. Most of the terms also have at least one other connotation which is more specifically related to hardware or computer operation. For example, responses to "object" might identify it as the name for the output of a compiler or as a data container in a program. Similarly, "interface" could be either explained as the connection between two pieces of hardware or as the logical definition for a piece of software (or both). Each respondent was given up to one point towards the software engineering orientation score for each definition given in that context and up to one point towards the hardware and computer operation score for each definition judged to be in that realm. Since some students gave multiple definitions, it was possible to receive credit for both.

The background questionnaire was designed to provide a profile of each student's breadth of experience in computer science as well as in possibly related areas. Specifically, it asked which computer languages were known, which were preferred and why, which natural languages were known well, and which were known somewhat. It also asked each student to name a favorite computer science course and explain why it was favored. Finally, it asked about artistic inclinations (music, writing, drama, art, etc.) and other liberal arts interests or abilities. One point was given for naming Pascal or another language which allows user type definitions as one's preferred language. Another point was given if the reason for this preference was the ability to declare and define distinct types within a program. A half point was given for each programming language known (up to eight languages), one point was given for each natural language known very well (up to four), and a half point was given for each additional natural language studied or known somewhat (up to four). A half point was given if the student's favorite course was in the language or software engineering fields and an additional half point was given if the choice was for reason of the topic rather than the instructor. Finally, a half point was given for each artistic or creative ability (up to four) and a quarter point was given for each related liberal arts interest, such as psychology, linguistics, or mathematics.

Course Data

The four other variables were derived from scores earned throughout and immediately after the course. The first two scores were from two exams, one given during the seventh week and one given in the fourteenth week. The first exam given in the control class emphasized the mechanics of the Ada language, which was emphasized in the first part of that course. A test of similar knowledge was given as the second exam in the experimental class since this material was presented later in that class. Conversely, the first exam in the experimental class tested the high-level concepts of Ada packaging and designing, while a similar test was used as the second exam in the control class. The last two scores were the grade on a final take-home design project and the composite course grade, both computed on one-hundred point scales.

Analysis

Confirmatory Findings

Several redundancies in the background data were shown to be internally consistent through significant correlations between the variables for both classes taken as a whole. (For this discussion, a cited correlation implies a correlation which would occur by chance less than 5% of the time; alpha less than 0.05.)

	SATV	SATM	CPAB	SE	HW&O	LANG
GPA	*	*		*		
SATV		*	*	*		*
SATM			*			
CPAB				*		
SE						*
HW&O						

* Indicates a positive correlation significant at the .05 level
(No significant negative correlations were found.)

GPA: grade point average
SATV: Scholastic Aptitude Test, verbal
SATM: Scholastic Aptitude Test, non-verbal
CPAB: Computer Programmers Aptitude Battery
SE: software engineering predisposition
HW&O: computer hardware and operations predisposition
LANG: breadth of language background

Figure 1.

	EXLO	EXHI	DSGN	CUM	section:
GPA	*		*	*	- control
			*	*	- experimental
	*	*	*	*	- both
SATV			*		- control
					- experimental
					- both
SATM	*	*		*	- control
					- experimental
					- both
CPAB	*	*		*	- control
	*			*	- experimental
	*	*		*	- both
SE	*	*	*	*	- control
		*		*	- experimental
	*		*	*	- both
HW&O					- control
			-*		- experimental
			-*		- both
LANG					- control
				*	- experimental
				*	- both

* Indicates a positive correlation significant at the .05 level
-* Indicates a negative correlation significant at the .05 level

GPA: grade point average
SATV: Scholastic Aptitude Test, verbal
SATM: Scholastic Aptitude Test, non-verbal
CPAB: Computer Programmers Aptitude Battery
SE: software engineering predisposition
HW&O: computer hardware and operations predisposition
LANG: breadth of language background
EXLO: exam on Ada details
EXHI: exam on design and packages
DSGN: design project, take-home final
CUM: cumulative course grade

Figure 2.

Grade point average correlated with both SAT verbal and non-verbal scores, which also correlated with each other. CPAB test scores also correlated with both SAT scores. Significant positive relationships also existed between grade point average and an orientation toward software engineering (term glossary), and between this orientation and both the CPAB score and the breadth of background. Breadth of background was related to SAT verbal but not to SAT non-verbal. See Figure 1 for a summary of the significant correlations between the background data for both classes together.

Confirmatory results were also found in the course variables, with every pair from the four variables correlating positively. Also, the author was reassured to learn that grade point average was a good predictor of each of the four course variables. Finally, it is worth pointing out that the CPAB test was also a good predictor of overall course grade, and the grades on the two exams, but not the grade on the final design project. (Although it might be the case that the CPAB test does not reveal abilities related to the design project, it also might be merely an indication that some individuals are good at tests no matter what the situation, while others are better at creative assignments.)

Experimental Comparison

No significant differences between the means of the variables measured for the two classes were found. Standard

deviations for most variables were fairly large in both sections indicating the high individual variance among students. However, there were some interesting differences between the two sections which show up in the correlations between background factors and course variables.

Figure 2 shows the significant correlations between the background data and the course data for each class individually and for both classes together. Some of the relationships indicated here have already been mentioned. In general, the best predictors of course success, in decreasing strength, were grade point average, CPAB test, software engineering predisposition, and breadth of language background. Notice that the latter two are quite simple to collect though they are somewhat less powerful.

There is some support for the original hypothesis that teaching design philosophy first is important. However, it appears that the background of the student is of even greater importance. Nevertheless, these data show how students with certain backgrounds perform better in one class than they do in the other.

Specifically, notice that to score well on the design problem given as part of the final exam, students in the control class (mechanics first) needed a predisposition to software engineering concepts. Those without this advantage did not perform as well on this task. This effect was absent from the experimental class (methodology first), possibly indicating that the presentation of the methodology first can compensate for a lack of prior knowledge.

Conversely, students in the experimental class who had the least predisposition to hardware and computer operation did better on the design task. Perhaps such an orientation can hamper acquisition of the important principles of design which are presented early in the experimental class when prior knowledge might be a more significant factor. This could possibly be an example of negative transfer which was discussed previously. For the classes taken together, both a predisposition toward software engineering and away from hardware and computer operation was associated with a high score on the design task.

Conclusions

Some overall guidelines to selecting a curriculum for a given student based only upon a predisposition to software engineering or to computer hardware and operation are suggested here. If a student scores well on the software engineering indicator (whether the hardware and operation indicator is high or low), he will do better in a mechanics-first class. However, if a student scores low on the computer operation and hardware indicator (whether the software engineering indicator is high or low), he will probably do better in a methodology-first class.

One additional overall message from these class distinctions seems to be that prior knowledge, predisposition, and orientation play a greater role during the first half of a semester-long course which explores new concepts than they do after the course is well under progress.

Further Study

Although this study did not find the overall indications that one particular method of teaching the Ada language was clearly superior to another, it did show that it is important to consider a student's background and prior skills before deciding upon an optimal curriculum. Although this may not always be practical on an individualized basis, it is expected that there are certain background effects that are common to large groups of industry programmers who must learn Ada in the near future. A better understanding of these large factors in the field will aid in the development of future Ada courses. The example provided by this study shows how a fairly simple measure can be used as a basis to make a first pass at selecting appropriate course materials.

It is expected that further study will continue in the area of education and technology transfer, not only with respect to Ada, but to an entire discipline of software development which must be advanced to a state of practice within industry.

Acknowledgements

The author would like to thank his advisor at the University of Maryland, Dr. Victor Basili for his consultation and support, Dr. Marv Zelkowitz for help in scheduling the Ada class, the General Electric Co. and specifically Drs. Pauline Jordan, Robert Farrell, and Larry Alexander for the opportunity to teach the course, and Dr. Elizabeth Kruesi Bailey for her encouragement and advice.

References

[1] J. Bailey, Transitioning to Ada: Exploiting a pilot development, in Proceedings of Using Ada, A New Era in Adaptable, Reliable Software, USPDI and AIAA, June 1983.

[2] A. Duncan, et al, Communication system design using Ada, in Proceedings of the Seventh International Conference on Software Engineering, IEEE, March 1984.

[3] Newsletters distributed as part of the University of Maryland - General Electric Co. joint study of a pilot Ada development.

[4] ANSI/MIL-STD-1815A, Military standard Ada programming language, January 1983 (revised reference manual for Ada).

[5] J.G.P. Barnes, Programming in Ada, Addison-Wesley, London, 1982.

[6] G. Lindzey, C. Hall, R. Thompson, Psychology, Worth, New York, 1978, p. 306.

[7] D. Krech, R. S. Crutchfield, Elements of Psychology, Knopf, New York, 1970, pp. 265 - 269.

[8] H. Sackmann, W.J. Erickson, E.E. Grant, Exploring experimental studies comparing online and offline programming performance, Communications of the ACM, Vol. 11, January 1968.

[9] S. B., Sheppard, E. Kruesi, The effects of the symbology and spatial arrangement of software specifications in a coding task. In Proceedings of Trends and Applications 1981: Advances in Software Technology. New York: IEEE, 1981.

[10] D. Slobin, Psycholinguistics, Scott, Foresman and Company, Glenview, 1971, p. 122.

[11] G. Y. DeNelsky, M. G. McKee, Prediction of computer programmer training and job performance using the AABP test, Personnel Psychology, Volume 27, 1974, p. 129.

END

DTIC

FILMED

4-86